Yousef Awad

Hash Table

1)

index:	0	1	2	3	4	5	6	7	8	9	10
value:	25	6		2		9			3	1	14

2)

index:	0	1	2	3	4	5	6	7	8	9	10
value:		14	6	2		9	25		3	1	

3)

index:	0	1	2	3	4	5	6	7	8	9	10
value:		26		2		9			^	1	

3→14→25

4

```
// Arup Guha
// 3/31/08
// Hash Table of words implemented using Linear Probing.

#include <stdio.h>
#include <string.h>

// Note: This constrains the limits of static memory allocation ...
#define MAX_SIZE 29
#define TABLE_SIZE 59999
// #define TABLE_SIZE 200

struct htable {
  char entries[TABLE_SIZE][MAX_SIZE + 1];
};

// Hash Table Functions.
void initTable(struct htable *h);
int hashvalue(char word[]);
void insertTable(struct htable *h, char word[]);
int searchTable(struct htable *h, char word[]);
void deleteTable(struct htable *h, char word[]);

int main() {

  char filename[MAX_SIZE + 1], temp[MAX_SIZE + 1];
  FILE *ifp;
  int numwords, i;
  struct htable mytable;
  int ans;

  // Get the file name.
  printf("What is the name of the dictionary file?\n");
  scanf("%s", &filename);
  ifp = fopen(filename, "r");

  fscanf(ifp, "%d", &numwords);

  // Read in all of the words from a file into the table.
  // This is only done once.
  printf("get here\n");
  initTable(&mytable);
```

```
43  printf("iniit table\n");
44
45  for (i = 0; i < numwords; i++) {
46    fscanf(ifp, "%s", temp);
47    insertTable(&mytable, temp);
48  }
49
50  // Allow the user to make changes to the hash table and search
      for words.
51  do {
52
53    printf("Do you want to 1)search word, 2) add word, 3) delete
      a word?\n");
54    scanf("%d", &ans);
55
56    // Search for a word.
57    if (ans == 1) {
58
59      printf("What word are you looking for?\n");
60      scanf("%s", temp);
61      if (searchTable(&mytable, temp))
62        printf("%s was found.\n", temp);
63      else
64        printf("%s was NOT found.\n", temp);
65    }
66
67    // Add a word.
68    else if (ans == 2) {
69
70      printf("What word do you want to add?\n");
71      scanf("%s", temp);
72      if (searchTable(&mytable, temp))
73        printf("%s was ALREADY in the table\n", temp);
74      else
75        insertTable(&mytable, temp);
76    }
77
78    // Delete a word.
79    else if (ans == 3) {
80
81      printf("What word do you want to delete?\n");
82      scanf("%s", temp);
83      deleteTable(&mytable, temp);
84    }
85
```

```
86    } while (ans < 4); // Not very user friendly, just quits for
        any number > 3.
87
88    return 0;
89  }
90
91  // Pre-condition: none
92  // Post-condition: Sets each entry in the hash table pointed to
        by h to the
93  //                 empty string.
94  void initTable(struct htable *h) {
95    int i;
96
97    // Our marker for an empty entry is the empty string.
98    for (i = 0; i < TABLE_SIZE; i++)
99      strcpy(h->entries[i], "");
100 }
101
102 // Pre-condition: none
103 // Post-condition: Calculates a hash value for word.
104 int hashvalue(char word[]) {
105
106   int i, sum = 0;
107
108   // Basically represents the value of word in base 128 (
        according to ascii
109   // values) and returns its value mod the TABLE_SIZE.
110   for (i = 0; i < strlen(word); i++)
111     sum = (128 * sum + (int)(word[i])) % TABLE_SIZE;
112
113   return sum;
114 }
115
116 // Pre-condition: h points to a valid hash table that IS NOT
        full.
117 // Post-condition: word will be inserted into the table h.
118 void insertTable(struct htable *h, char word[]) {
119
120   int hashval;
121   hashval = hashvalue(word);
122
123   // Here's the linear probing part.
124   /* while (strcmp(h->entries[hashval], "") != 0)
125    *       hashval = (hashval+1)%TABLE_SIZE;
126    */
```

```
127   int i = 0;
128   while (strcmp(h->entries[(hashval + i * i) % TABLE_SIZE], "")
       != 0) {
129     i++;
130   }
131   strcpy(h->entries[(hashval + i * i) % TABLE_SIZE], word);
132 }
133 
134 // Pre-condition: h points to a valid hash table.
135 // Post-condition: 1 will be returned iff word is stored in the
       table pointed to
136 //                 by h. Otherwise, 0 is returned.
137 int searchTable(struct htable *h, char word[]) {
138 
139   int hashval;
140   hashval = hashvalue(word);
141 
142   // See what comes first, the word or a blank spot.
143   int i = 0;
144   while (strcmp(h->entries[(hashval + i * i) % TABLE_SIZE], "")
       != 0 &&
145          strcmp(h->entries[(hashval + i * i) % TABLE_SIZE], word
       ) != 0) {
146     i++;
147   }
148 
149   // The word was in the table.
150   if (strcmp(h->entries[(hashval + i * i) % TABLE_SIZE], word)
       == 0)
151     return 1;
152 
153   // It wasn't.
154   return 0;
155 }
156 
157 // Pre-condition: h points to a valid hash table.
158 // Post-condition: deletes word from the table pointed to by h,
       if word is
159 //                 stored here. If not, no change is made to the
       table pointed
160 //                 to by h.
161 void deleteTable(struct htable *h, char word[]) {
162 
163   int hashval;
164   hashval = hashvalue(word);
```

```
165
166   // See what comes first, the word or a blank spot.
167   int i = 0;
168   while (strcmp(h->entries[(hashval + i * i) % TABLE_SIZE], "")
       != 0 &&
169          strcmp(h->entries[(hashval + i * i) % TABLE_SIZE], word
       ) != 0) {
170     i++;
171     if ((hashval + i * i) % TABLE_SIZE) {
172       return;
173     }
174   }
175
176   // Reset the word to be the empty string.
177   if (strcmp(h->entries[(hashval + i * i) % TABLE_SIZE], word)
       == 0) {
178     strcpy(h->entries[(hashval + i * i) % TABLE_SIZE], "");
179   }
180
181   // If we get here, the word wasn't in the table, so nothing is
       done.
182 }
```